CLASSIFICATION OF ASL WORDS USING CNNS

Authors
The Probables
{kaviaras, kjayandr, naveenud}@buffalo.edu

1 Introduction

This dataset is provided by our current research project, which utilize 20 kHz acoustic sensing to sense ASL gestures. All the 10 ASL words perform by 5 subjects. All images are generated by using the short-time Fourier transform (STFT) to calculate a spectrogram as the feature representation of the reflected near-ultrasound waves. Based on the Doppler effect, sign language gestures, including both hands and arms, will cause phase and frequency changes of the reflected sonic wave. The spectrogram contains information in both frequency and time domains. The spectrogram is also defined as the Power Spectral Density of the function:

$$\text{spectrogram}\{x(t)\}(\tau,\omega) \equiv |X(\tau,\omega)|^2 = \left| \sum_{n=-\infty}^{\infty} x[n]\omega[n-m]e^{-j\omega n} \right|^2 \tag{1}$$

where $x[n]$ is input signal, and $\omega[n-m]$ represents the overlapping Kaiser window function with an adjustable shape factor β that improves the resolution and reduces the spectral leakage close to the sidelobes of the signal. The coefficients of the Kaiser window are computed as:https://www.overleaf.com/project/627eaa6607dfe60d3b85ba38

$$\omega[n] = \frac{I_0\left(\beta\sqrt{1-\left(\frac{n-N/2}{N/2}\right)^2}\right)}{I_0(\beta)}, 0 \le n \le N \tag{2}$$

The given dataset contains images that are generated by using the short-time Fourier transform in calculating the spectrogram as a feature representation of the reflected near ultrasound waves. The given spectrogram contains information regarding the frequency as well as the time domains. Each of the images belongs to one of the 10 classes and the aim is to classify the images correctly into their respective categories. The 10 classes are big, choose, cold, family, hello, I, mirror, need, nice and small.

This dataset has a training set of 5,000 examples, and a test set of 1,000 examples.

2 Loading the dataset

To load the dataset that is given in the problem, we use Keras ImageDataGenerator that can take these given folders as arguments. The Keras ImageDataGenerator's flow_from_directory() method takes a path of a directory and generates batches of augmented data.

```
# load the file from our dataset including training, validation and testing part
img_width, img_height = 224, 224

#TO-DO change the path to local path
train_data_dir      = 'pictures/train'
validation_data_dir = 'pictures/val'
test_data_dir       = 'pictures/test'

target_size = (img_width, img_height)
```

```
epochs = 50
batch_size = 16
num_classes = 10

# # this is a generator that will read pictures found in
# # subfolers of 'data/train', and indefinitely generate
# # batches of augmented image data
# Normalizing the data
train_datagen = ImageDataGenerator(rescale=1./255)
valid_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# Image generators to read the data from the directories
train_generator = train_datagen.flow_from_directory(train_data_dir,
                                                    target_size=target_size)
valid_generator = valid_datagen.flow_from_directory(validation_data_dir,
                                                    target_size = target_size)
test_generator  = test_datagen.flow_from_directory(test_data_dir,
                                                    target_size = target_size)

# please print the number of samples in each folder
print('Number of training samples : ' + str(train_generator.samples))
print('Number of validation samples : ' + str(valid_generator.samples))
print('Number of test samples : ' + str(test_generator.samples))
```

Number of training samples : 4176
Number of validation samples : 1392
Number of test samples : 1392

3 Implementing the Convolutional Neural Network

A Convolutional Neural Network is a Deep Learning algorithm which can take in an input image, learn weights and bias to various aspects/objects in the image and be able to differentiate one from the other.

3.1 Building a Convolutional Neural Network without regularization

We built our neural network using 3 convolutional layers with 32, 64 and 128 filters, and 2 fully connected dense layers with 128 and 256 units. We used Relu activation function for the three convolutional layers and the two fully connected dense layers. The final output layer has 10 units and uses Softmax as the activation function which gives output as a probability distribution in the range of (0,1) and the outputs from all the units sum up to 1. To predict a new unseen image, we can use model.predict() with the unseen image as an input. The output will be an array of probability distribution values of the image belonging to the 10 classes. We can then use numpy's argmax function to find the class that the image belongs to.

```
# build conv2D CNN model, be careful with softmax and output layers is 10
from keras.callbacks import EarlyStopping
# define the input shape of Convolutional Neural Network
# Your Code HERE
input_shape = (img_height, img_width, 3)
num_classes = 10

# define the Convolutional Neural Network
# Your Code HERE
model = Sequential([
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
        layers.MaxPooling2D(pool_size=(2,2)),
        layers.Conv2D(64, kernel_size=(3,3), activation='relu'),
        layers.MaxPooling2D(pool_size=(2,2)),
        layers.Conv2D(128, kernel_size=(3,3), activation='relu'),
        layers.MaxPooling2D(pool_size=(2,2)),
```

```
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dense(256, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])

model.summary()
```

Figure 1: Model summary for CNN without regularization

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 222, 222, 32)      896

max_pooling2d (MaxPooling2D  (None, 111, 111, 32)     0
)

conv2d_1 (Conv2D)            (None, 109, 109, 64)      18496

max_pooling2d_1 (MaxPooling  (None, 54, 54, 64)       0
2D)

conv2d_2 (Conv2D)            (None, 52, 52, 128)       73856

max_pooling2d_2 (MaxPooling  (None, 26, 26, 128)      0
2D)

flatten (Flatten)            (None, 86528)             0

dense (Dense)                (None, 128)               11075712

dense_1 (Dense)              (None, 256)               33024

dense_2 (Dense)              (None, 10)                2570

=================================================================
Total params: 11,204,554
Trainable params: 11,204,554
Non-trainable params: 0
_________________________________________________________________
```

Compiling and training the above model

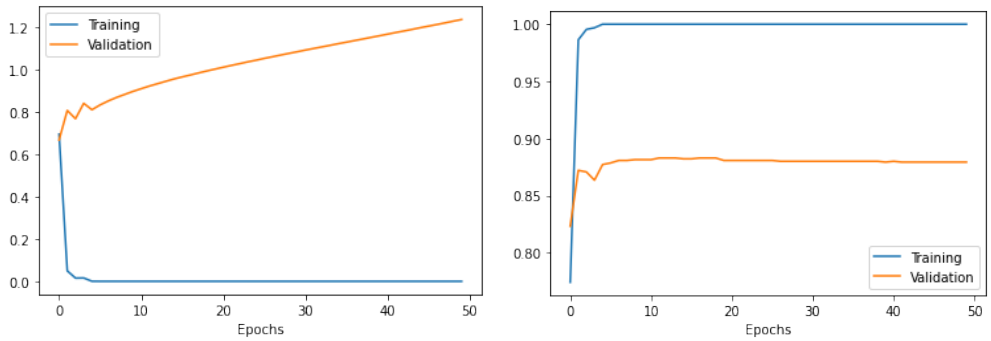
```
# Compiling the model and training including the files of compile and fit
#Your code
optimizer = keras.optimizers.Adam(learning_rate=0.0003)
model.compile(loss="categorical_crossentropy", optimizer=optimizer, metrics=["
                                        accuracy"])

#Train the model with training and validation set
history = model.fit(train_generator,
                    validation_data = valid_generator,
                    epochs=epochs,
                    batch_size=batch_size)
```

Code for plotting the loss for training and validation data.

```
#Plotting loss trend
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])

plt.legend(['Training', 'Validation'])
plt.xlabel('Epochs')
```

Code for plotting the accuracy for training and validation data.

```
# Plotting accuracy trend
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['Training', 'Validation'])
```



(a) Loss trend for training vs validation (b) Accuracy trend for training vs validation

Figure 2: Plotting graph for loss and accuracy trend for CNN without regularization

```
plt.xlabel('Epochs')
```

Training the model for 50 epochs, we get the following loss and accuracy which is plotted in the graph below. Refer 2

To test the model we use the following code

```
# Test the model on a testing dataset
score = model.evaluate(test_generator)
print('Loss on test dataset is ' + str(score[0]))
print('Accuracy on test dataset is ' + str(score[1]))
```

We got accuracy of 83.12%. We also tried models with different number of convolutional layers, different number of filters and without fully connected layers. None of the models achieved an accuracy that surpassed 80%.

The model does overfit during the training phase. Overfitting in a neural network occurs when the model fits too well to the training set. It then becomes difficult for the model to generalize to new examples that were not in the training set. This could occur when the model is too deep or there isn't enough training data. Although our implementation is not that deep with three convolutional layers and two fully connected layers, there was some amount of overfitting.

There are several methods to address overfitting like augmenting the image data, introducing dropout layers, having skip connections, imposing regularization. We applied L1 and L2 kernel regularizers with a regularization factor of 0.01. Regularizers allow you to apply penalties on layer parameters or layer activity during optimization. These penalties are summed into the loss function that the network optimizes.

3.2 Building a Convolutional Neural Network with L1 regularization

In L1 (Lasso) regularization, while taking derivative of the cost function it will estimate around the median of the data.Therefore, to minimize the loss function, we should try to estimate a value that should lie at the mid of the data distribution.

In order to prevent overfitting, we used L1 regularization to the existing model. Keras allows three types of applying regularization to the layers - kernel regularizers, bias regularizers and activity regularizers. We opted to use **L1 kernel regularizers** on the three convolutional layers with a regularization factor of 0.01.

```
#Adding L1 regularization to the existing model
from keras.regularizers import l1

l1_model = Sequential([
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation='relu',
                                                kernel_regularizer=l1(0.01)),
        layers.MaxPooling2D(pool_size=(2,2)),
        layers.Conv2D(64, kernel_size=(3,3), activation='relu', kernel_regularizer
                                                =l1(0.01)),
        layers.MaxPooling2D(pool_size=(2,2)),
```

```
        layers.Conv2D(128, kernel_size=(3,3), activation='relu',
                                                kernel_regularizer=l1(0.01)),
        layers.MaxPooling2D(pool_size=(2,2)),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dense(256, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])

l1_model.summary()
```

Figure 3: Model summary for CNN with L1 regularization

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 222, 222, 32)      896

max_pooling2d_3 (MaxPooling  (None, 111, 111, 32)      0
2D)

conv2d_4 (Conv2D)            (None, 109, 109, 64)      18496

max_pooling2d_4 (MaxPooling  (None, 54, 54, 64)        0
2D)

conv2d_5 (Conv2D)            (None, 52, 52, 128)       73856

max_pooling2d_5 (MaxPooling  (None, 26, 26, 128)       0
2D)

flatten_1 (Flatten)          (None, 86528)             0

dense_3 (Dense)              (None, 128)               11075712

dense_4 (Dense)              (None, 256)               33024

dense_5 (Dense)              (None, 10)                2570

=================================================================
Total params: 11,204,554
Trainable params: 11,204,554
Non-trainable params: 0
_________________________________________________________________
```

Compiling and training the above model

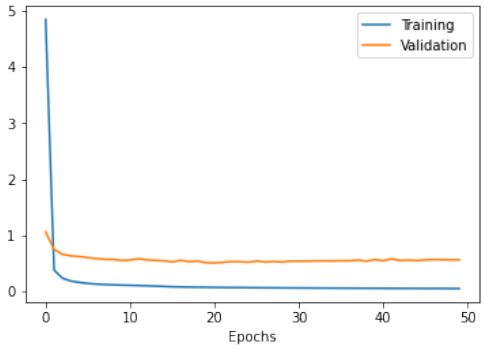
```
# Compiling the model and training including the files of compile and fit
l1_model.compile(loss="categorical_crossentropy", optimizer=optimizer, metrics=["
                                        accuracy"])

#Train the model with training and validation set
l1_history = l1_model.fit(train_generator,
                    validation_data = valid_generator,
                    epochs=epochs,
                    batch_size=batch_size)
```

Code for plotting the loss for training and validation data.

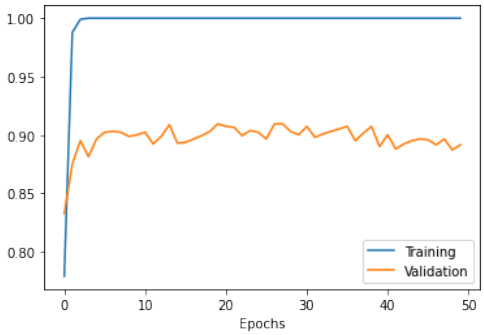
```
#Plotting loss trend
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])

plt.legend(['Training', 'Validation'])
plt.xlabel('Epochs')
```

Code for plotting the accuracy for training and validation data.

```
# Plotting accuracy trend
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
```



(a) Loss trend for training vs validation



(b) Accuracy trend for training vs validation

Figure 4: Plotting graph for loss and accuracy trend for CNN with L1 regularization

```
plt.legend(['Training', 'Validation'])
plt.xlabel('Epochs')
```

Training the model for 50 epochs, we get the following loss and accuracy which is plotted in the graph below. Refer 4

To test the model we use the following code

```
# Test the model on a testing dataset
score = model.evaluate(test_generator)
print('Loss on test dataset is ' + str(score[0]))
print('Accuracy on test dataset is ' + str(score[1]))
```

We got accuracy an accuracy of 87.5% and it can also be seen from the accuracy and loss trend of the training process that the overfitting has reduced significantly.

3.3 Building a Convolutional Neural Network with L2 regularization

In L2 (Ridge) regularization, while calculating the loss function in the gradient calculation step, the loss function tries to minimize the loss by subtracting it from the average of the data distribution.

We also used L2 regularization to the existing model. We opted to use **L2 kernel regularizers** on the three convolutional layers with a regularization factor of 0.01.

```
from keras.regularizers import l2

#Adding L2 regularization to the existing model
l2_model = Sequential([
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation='relu',
                                                kernel_regularizer=l2(0.01)),
        layers.MaxPooling2D(pool_size=(2,2)),
        layers.Conv2D(64, kernel_size=(3,3), activation='relu', kernel_regularizer
                                                =l2(0.01)),
        layers.MaxPooling2D(pool_size=(2,2)),
        layers.Conv2D(64, kernel_size=(3,3), activation='relu',
        kernel_regularizer=l2(0.01)),
        layers.MaxPooling2D(pool_size=(2,2)),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dense(256, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])

l2_model.summary()
```

Figure 5: Model summary for CNN with L2 regularization

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 222, 222, 32)      896

 max_pooling2d_6 (MaxPooling  (None, 111, 111, 32)     0
 2D)

 conv2d_7 (Conv2D)           (None, 109, 109, 64)      18496

 max_pooling2d_7 (MaxPooling  (None, 54, 54, 64)       0
 2D)

 conv2d_8 (Conv2D)           (None, 52, 52, 64)        36928

 max_pooling2d_8 (MaxPooling  (None, 26, 26, 64)       0
 2D)

 flatten_2 (Flatten)         (None, 43264)             0

 dense_6 (Dense)             (None, 128)               5537920

 dense_7 (Dense)             (None, 256)               33024

 dense_8 (Dense)             (None, 10)                2570

=================================================================
Total params: 5,629,834
Trainable params: 5,629,834
Non-trainable params: 0
_________________________________________________________________
```

Compiling and training the above model

```
l2_model.compile(loss="categorical_crossentropy", optimizer=optimizer, metrics=["
                                        accuracy"])

#Train the model with training and validation set
l2_history = l2_model.fit(train_generator,
                    validation_data = valid_generator,
                    epochs=epochs,
                    batch_size=batch_size)
```

Code for plotting the loss for training and validation data.

```
#Plotting loss trend
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])

plt.legend(['Training', 'Validation'])
plt.xlabel('Epochs')
```

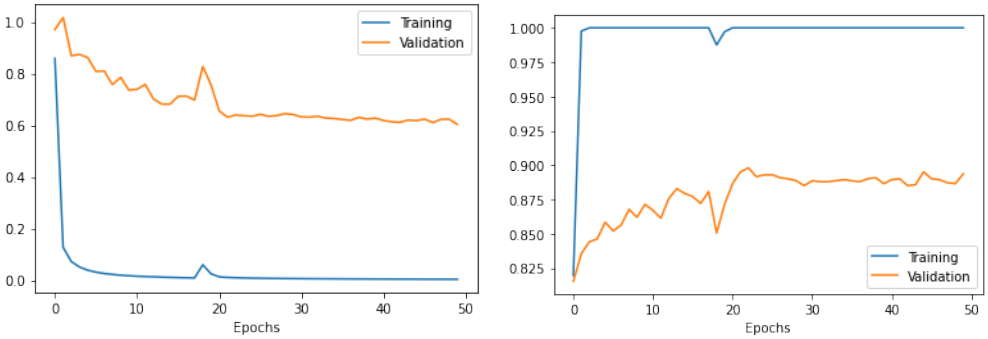
Code for plotting the accuracy for training and validation data.

```
# Plotting accuracy trend
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['Training', 'Validation'])
plt.xlabel('Epochs')
```

Training the model for 50 epochs, we get the following loss and accuracy which is plotted in the graph below. Refer 6

To test the model we use the following code

```
# Test the model on a testing dataset
score = model.evaluate(test_generator)
print('Loss on test dataset is ' + str(score[0]))
print('Accuracy on test dataset is ' + str(score[1]))
```



(a) Loss trend for training vs validation (b) Accuracy trend for training vs validation

Figure 6: Plotting graph for loss and accuracy trend for CNN with L2 regularization

We got an accuracy of 88.43%, which is slightly better than L1 regularization. L1 tends to shrink coefficients to zero whereas L2 tends to shrink coefficients evenly which makes L2 regularization useful especially when there are collinear or codependent features.

In L2 regularization, the weights obtain a value closer to 0 , this would ensure that the hidden neurons with smaller weights do not have an impact on the model. This subsequently reduces the complexity of the neural network while still ensuring that a higher accuracy is obtained. https://www.overleaf.com/project/627eaa6607dfe60d3b85ba38

4 ResNet-50 convolutional neural network

ResNet-50, a variation of Resnet is a convolutional neural network that is 50 layers deep. The network has learned rich feature representations for a wide range of images and an image input size of 224-by-224. The Residual learning helps in overcoming the complications that are usually caused due to very deep neural networks.It implements the concept of skipping connections in it's model. Skip connections combine the outputs of prior layers with the outputs of stacked layers, allowing for considerably deeper network training than before. This method also solves the problem of disappearing gradients by creating an alternate path for the gradient to follow. They also allow the model to learn an identity function. This guarantees that the model's upper levels perform equally well as the lower layers.Hence residual blocks make learning identity functions much easier for the layers. As a result, ResNet boosts the performance of deep neural networks with additional neural layers while lowering the error rate.

```
from keras.applications.resnet import ResNet50

# Pretrained Resnet50 model
base_model = ResNet50(include_top = False, weights='imagenet')
optimizer = keras.optimizers.Adam()
trained_models = []
#Building a model by freezing selected layers in the pretrained model and adding a
                                        fully connected layer to fit the given
                                       dataset
def build_model(start, end, base_model, input_shape, num_classes):
  print("Freezing layers from {} to {}".format(start, end))
  for layer in base_model.layers[start:end]:
    layer.trainable = False
  resnet = Sequential([keras.Input(shape = input_shape),
                       base_model,
                       layers.Flatten(),
                       layers.Dense(num_classes, activation='softmax')])
  print(resnet.summary())
  return resnet

#Training the model using images from the training dataset and validation dataset
def train_model(resnet_model, optimizer, train_generator, valid_generator,
                                        batch_size, epochs):
```

```
  resnet_model.compile(loss = 'categorical_crossentropy', optimizer = optimizer,
                                                  metrics=['accuracy'])
  resnet_history = resnet_model.fit(
    x=train_generator,
    validation_data=valid_generator,
    batch_size=batch_size,
    epochs=epochs)
  trained_models.append(resnet_history)
  return resnet_history

#Plotting accuracy and loss trends of the training process
def plot_trends(history, metric1, metric2):
  plt.plot(history.history[metric1])
  plt.plot(history.history[metric2])
  plt.legend(['Training', 'Validation'])
  plt.xlabel('Epochs')
  plt.show()

#Evaluating the built model on the test dataset
def evaluate(model, generator):
  score = model.evaluate(generator)
  print('Loss on test dataset is ' + str(score[0]))
  print('Accuracy on test dataset is ' + str(score[1]))
  print('*'*40)

#Defining the freeze layers and calling the above function to train and predict
                                              different models
config = [(0, 30), (30, 60), (60, 90)]
for i in config:
    start = i[0]
    end = i[1]
    model = build_model(start, end, base_model, input_shape, num_classes)
    history = train_model(model, optimizer, train_generator, valid_generator,
                                                batch_size, epochs)
    evaluate(model, test_generator)

for i in trained_models:
    plot_trends(i, 'loss', 'val_loss')
    plot_trends(i, 'accuracy', 'val_accuracy')
```

Figure 7: Model summary for freezing layers from 0 to 30

```
Freezing layers from 0 to 30
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, None, None, 2048)  23587712

 flatten_5 (Flatten)         (None, 100352)            0

 dense_11 (Dense)            (None, 10)                1003530

=================================================================
Total params: 24,591,242
Trainable params: 24,365,642
Non-trainable params: 225,600
_________________________________________________________________
```

Training the model for 50 epochs, we get the following loss and accuracy which is plotted in the graph below.

Figure 8: Model summary for freezing layers from 30 to 60

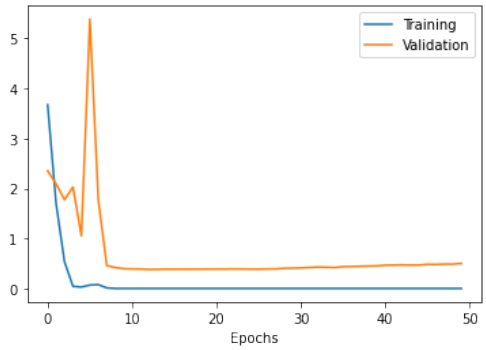
```
Freezing layers from 30 to 60
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, None, None, 2048)  23587712

 flatten_6 (Flatten)         (None, 100352)            0

 dense_12 (Dense)            (None, 10)                1003530

=================================================================
Total params: 24,591,242
Trainable params: 23,649,802
Non-trainable params: 941,440
_________________________________________________________________
```

Figure 9: Model summary for freezing layers from 60 to 90

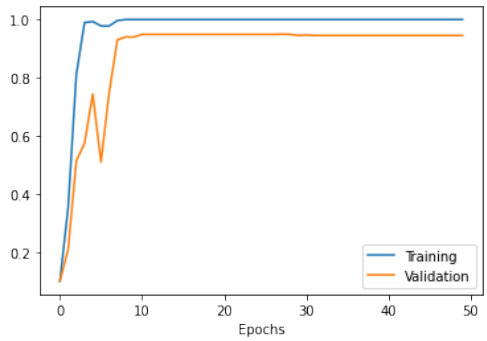
```
Freezing layers from 60 to 90
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, None, None, 2048)  23587712

 flatten_7 (Flatten)         (None, 100352)            0

 dense_13 (Dense)            (None, 10)                1003530

=================================================================
Total params: 24,591,242
Trainable params: 21,575,178
Non-trainable params: 3,016,064
_________________________________________________________________
```

Name	Accuracy
Resnet-50 model with freezing 0 to 30 layers	0.92
Resnet-50 model with freezing 30 to 60 layers	0.83
Resnet-50 model with freezing 60 to 90 layers	0.83

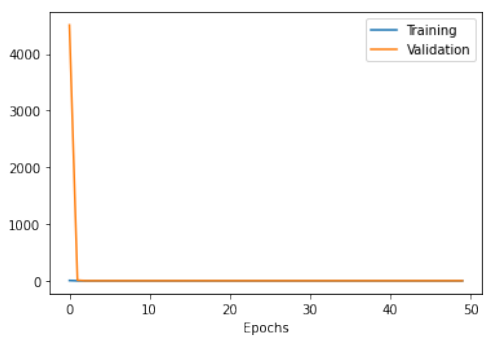
Table 1: Accuracies for different Resnet-50 models



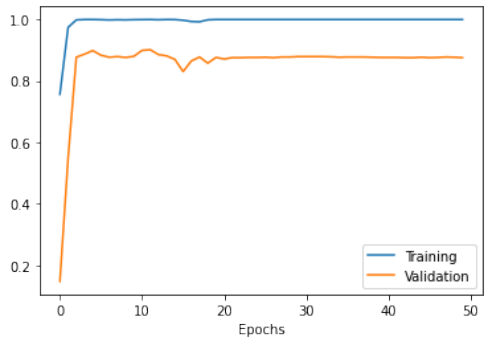
(a) Loss trend for training vs validation with freezing 0:30 layers



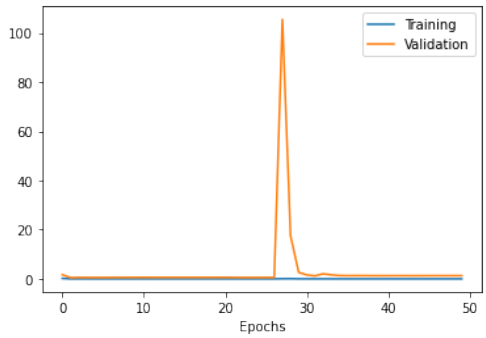
(b) Accuracy trend for training vs validation with freezing 0:30 layers



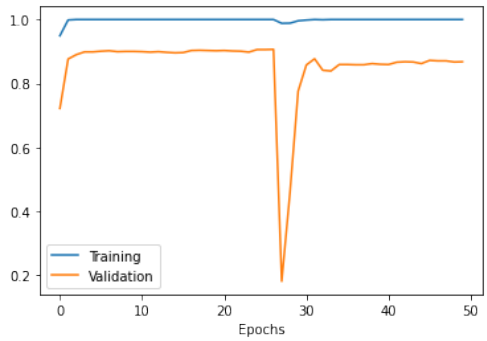
(c) Loss trend for training vs validation with freezing 30:60 layers



(d) Accuracy trend for training vs validation with freezing 30:60 layers



(e) Loss trend for training vs validation with freezing 60:90 layers



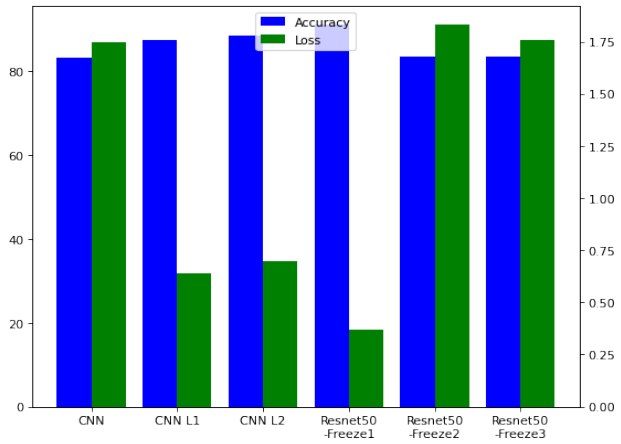
(f) Accuracy trend for training vs validation with freezing 60:90 layers

Figure 10: Plotting graph for loss and accuracy trend on different Resnet-50 configurations

5 Comparison of different models

The following figure compares the accuracies and losses on the test dataset for the various neural network models.

Figure 11: Comparison of accuracies and losses



References

[1] https://www.tensorflow.org/api_docs/python/tf/keras

[2] Slides from Kenny's lectures

[3] https://www.kaggle.com/code/cdeotte/how-to-choose-cnn-architecture-mnist/notebook

[4] https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/vision/ipynb/mnist_convnet.ipynbscrollTo=FF2A